Matrix Multiplication Algorithm Selection with Support Vector Machines

Omer Spillinger
David Eliahu
Armando Fox
James Demmel

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2015-29
http://www.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-29.html

May 1, 2015

Matrix Multiplication Algorithm Selection with Support Vector Machines

Omer Spillinger*†, David Eliahu§†, Armando Fox‡†, and James Demmel¶

***Abstract*—We present a machine learning technique for the *algorithm selection* problem, specifically focusing on algorithms for dense matrix multiplication. Dense matrix multiplication is a core component of many high-performance computing and machine learning algorithms [1], but the performance of matrix multiplication algorithms can vary significantly based on input parameters and hardware architecture. We build performance models for multiple machines using support vector machines (SVMs) [2] and show that only a sparse exploration of the input space is sufficient to accurately predict the best choice of algorithm over a wide range of possible inputs. We find that by using this classifier-based approach to choose the best algorithm to use at runtime, we are able to achieve as much as a 26% increase in average performance over choosing a single algorithm a priori. This is within 1.5% of the performance possible with a perfect algorithm selector.**

I. Introduction

Algorithm designers produce an ever-increasing set of techniques for solving an ever-wider set of problems. The factors that influence an algorithm's performance are complicated, and building an accurate analytical model given input parameters and hardware platform can require extensive domain expertise. Performance can vary significantly from system to system due to differences in factors such as memory hierarchy and number of processors. For these reasons, selecting the optimal algorithm for a particular problem has become a challenge in itself, particularly in fields where performance is critical such as scientific computing and machine learning.

When there exist multiple algorithms that solve the same problem, the typical approach is to always use the algorithm with the best average performance on a given problem distribution. However, this is problematic because there are many algorithms that are uncompetitive on average, but are ideal for some problem instances with certain sets of inputs [3].

The benefits of algorithm selection are twofold: performance can be optimized for all problem instances, and algorithm designers can focus on developing separate algorithms that target different portions of the problem space. A solution to the algorithm selection problem would enable the development of libraries that could intelligently choose the optimal algorithm for a particular set of inputs. Users would be oblivious to the underlying algorithmic implementation, confident that the library will choose the proper algorithm for the given inputs and hardware.

An analytical approach to this problem—developing a theoretical model of algorithm performance *a priori*—is not sensible in most cases. Such models are complex, and building one requires significant effort and domain expertise. More importantly, this approach does not generalize since performance models are dependent on both the algorithm and the performance characteristics of a particular machine.

One could also take an empirical approach: simply measure how algorithms perform on a given machine under different inputs. A naïve solution would be to exhaustively search the space of possible inputs; this is intractable in the general case, and even in situations where the range of possible inputs is finite, exhaustive search is time consuming.

The complexity of performance modeling lends itself to a machine learning approach. Rather than building a model of machine performance that depends on factors such as the memory hierarchy and number of cores, we train a classifier on a set of training examples to predict the best algorithm to use for a particular machine. In contrast to exhaustive search, a classifier-based approach need not explore a large portion of the search space. The user can make an explicit tradeoff between classifier accuracy and training time.

In this work, we present and evaluate an SVM-based algorithm selector for the problem of dense matrix multiplication. We choose this problem for several reasons. First, matrix multiplication is an important component of many scientific computing and machine learning tasks, and it is often the performance bottleneck for those tasks [1]. Thus, even relatively small performance gains for matrix multiplication can translate into significant savings for large tasks. Second, matrix multiplication is easily parameterizable—in general, performance is dictated by the dimensions of the input matrices. Finally, a plethora of matrix multiplications algorithms exist in the literature. We explore two implementations: the Intel Math Kernel Library (MKL) algorithm [4], and the communication-avoiding recursive algorithm, CARMA [5]. The CARMA and MKL implementations are substantially different,

*omers88@eecs.berkeley.edu
†EECS Department, UC Berkeley, Berkeley, CA 94720
§deliahu@eecs.berkeley.edu
‡fox@cs.berkeley.edu
¶demmel@cs.berkeley.edu, CS Division and Mathematics Department, UC Berkeley, Berkeley, CA 94720

which leads to noticeable variation in relative performance across different machines.

We evaluate our algorithm selector on a set of rectangular matrix multiplication problems ranging in size from 64 × 64 to 4096 × 4096 (a state space of over 65 billion possible inputs) and on three hardware platforms. All problems we explore fit in DRAM on all of our machines. We define accuracy as the fraction of times the faster algorithm is chosen by our classifier. Despite wide variations in performance across multiple machines for both CARMA and MKL, we show that our classifier achieves accuracy ranging from 85% to 87%.

Contrary to standard classification problems, our key measure of success is not classification accuracy—rather, we aim to maximize the average performance improvement of algorithm selection over preselecting a single algorithm. Higher classifier accuracy does not necessarily imply higher average performance because for problem sizes where the two algorithms have similar performance, choosing the faster algorithm does not have a significant effect on the average performance. Our technique yields performance that, in the worst case, falls within 1.5% of a perfect algorithm selector. Compared to choosing a single algorithm in advance, our approach increases average performance by up to 11% versus CARMA alone and 26% versus MKL alone.

II. Contributions

In this work we offer a methodology for solving the algorithm selection problem. We build a model using SVM to predict relative algorithm performance at runtime. In addition, we weight training datapoints in order to build a model that correctly classifies a higher percentage of the problem instances with the most significant performance variations. We show the potential of this approach for linear algebra algorithms, as we achieve up to a 26% performance improvement over selecting a single algorithm in advance.

III. Related Work

The idea of using machine learning techniques to improve performance on complex architectures is not new. In fact, the *algorithm selection problem* has been defined as early as 1976 [6]. Increasingly complicated machine architectures and compilers motivate a shift from theoretical to empirical approaches. However, most literature on this topic focuses on autotuning for templated code optimization problems using regression models [7]. Our approach is not limited to autotuning within an algorithm template, as we are more interested in performance variations between paradigmatically different algorithms (such as industry-standard linear algebra algorithms and their communication-avoiding counterparts) [5].

One way to tackle algorithm selection involves distilling problems into a set of features from which to model runtime performance. After collecting performance data from sample problems, regression may be used to learn a real-valued function of the features [3]. This process can be repeated for an arbitrary number of algorithms for a given type of problem. The best algorithm to select would be the one with the lowest predicted runtime based on the available models. Unlike our SVM [2] approach, which uses relative performance to determine the optimal algorithm, this approach allows for portable models without other algorithms as dependencies. In other words, our approach builds models that are only effective for selecting an algorithm from the set of algorithms that were used in training the classifier. However, high prediction accuracy requires sufficient domain expertise to define appropriate features, e.g. memory access patterns and parallelism characteristics. Our SVM approach requires no such domain expertise.

Autotuning has also been used for solving $\mathcal{NP}$-complete problems such as propositional satisfiability (SAT). The algorithms required for these problems are highly complex. Therefore, empirical studies are far more practical than theoretical analysis for modeling their performance. There is no single algorithm that is optimal for all SAT problem instances. Together, the infeasibility of theoretical models and the performance variations between SAT algorithms motivate building empirical hardness models (computationally inexpensive predictors of algorithm runtime) [8].

The algorithm selection problem can also be modeled as a Markov decision process (MDP). Different algorithms that solve a given problem represent actions, and state transitions occur when recursive calls are made. Cost is derived from the time required to solve a problem. The objective is to determine a policy that minimizes expected execution time. If a recursive algorithm generates multiple subproblems, the MDP is cloned for each of the state transitions [9]. This research focuses on using reinforcement learning in order to make optimal algorithm selection decisions at each recursive call. The machine learning is tightly coupled with the algorithm's execution whereas SVM treats algorithms as black boxes.

Another relevant research topic is the improvement of exhaustive search techniques. Heuristics may be used for terminating exhaustive search early if near-optimal implementations are found. In addition, run-time decision rules can be used to select fast implementations based on run-time input [10], [11]. Although this approach is similar to ours, it explores a two-dimensional rather than three-dimensional space of tuning parameters and tunes variables within a single algorithmic template while we focus on algorithm selection.

Algorithm selection is highly relevant to compiler research. The PetaBricks programming language allows users to express algorithm selection at the language level [12]. Similarly, OpenTuner [13], a general framework for program autotuning, supports algorithm selection. The framework demonstrates effective usage of ensembles of search techniques to explore complex search spaces. Our

TABLE I: Machines used in this study.

Machine	Cores	Threads	CPU Type
Hopper	24	24	AMD 'MagnyCours'
Emerald	32	64	Intel Xeon X7560
Boxboro	40	80	Intel Xeon E7-4860

SVM approach for algorithm selection could be integrated into the OpenTuner project in order to enhance its autotuning capabilities.

IV. Data

Our project takes an empirical approach to the algorithm selection problem. Thus, our first step is to collect the data from which to build our model.

A. Generating Data

We featurize matrix multiplications based on the dimensions of the input matrices: m, k, and n. These features represent the size of each dimension of a matrix multiplication of the form $A \times B$, where A is an $m \times k$ matrix and B is a $k \times n$ matrix. We generate random dense matrices with real-valued floating point numbers. The matrices range in size from 64×64 to 4096×4096, and we vary each dimension in 10 evenly-spaced increments across this range. Note that this means we generate a variety of rectangular matrices, not just square ones.

This step results in 1000 matrix multiplications (i.e., combinations of m, k, and n). We run each of these matrix multiplications using both of our algorithms under test (MKL and CARMA) and record the performance of each multiplication in billions of flops per second (Gflops). We repeat each multiplication fifteen times and compare the max performance for each algorithm (because max is the best measure of an algorithm's peak performance on a given machine)[1]. We performed this process on three separate machines, building three independent models. The machines we used are shown in Table I.

To multiply the matrices, we wrote a custom timing mechanism in C to measure the performance of MKL and CARMA. Our program allocates three matrices (A, B, and C), initializes them with random floating-point numbers between -1 and 1, and warms the cache before each trial. To ensure that the computation takes sufficient time for accurate measurements, we multiply enough matrices such that the total time of all multiplications is at least 0.2 seconds. We then divide the total time by the number of multiplications to calculate the time for a single matrix multiplication. To compute Gflops, we divide $2^{-9} * m * k * n$ by the time for a single multiplication[2].

[1]We found that fifteen iterations is the number sufficient to measure maximum performance on our machines.

[2]Each entry of the resulting $m \times n$ matrix C requires $2k$ operations (k multiplies and k adds), hence this total for Gflops.

B. Limitations

Our data has a few limitations. First, our largest matrix is 4096×4096. While this is a large matrix, it still fits in DRAM on all of our machines; we did not explore extremely large matrix sizes, though we believe our technique is general enough to scale to those as well. Secondly, some of our machines exhibit significant variability across multiple multiplications of the same dimensions. The average coefficients of variation were 0.08, 0.05, 0.11 for Hopper, Emerald, and Boxboro, respectively. This variability is a result of how the matrices are allocated across NUMA regions, which we do not control. To accurately evaluate how fast the multiplication would run with an ideal data layout, we took the max performance measurement over fifteen trials. Our classification results were strong (see Section VII), so we do not believe that this approximation had a major impact on our evaluation or the significance of our results.

V. Training

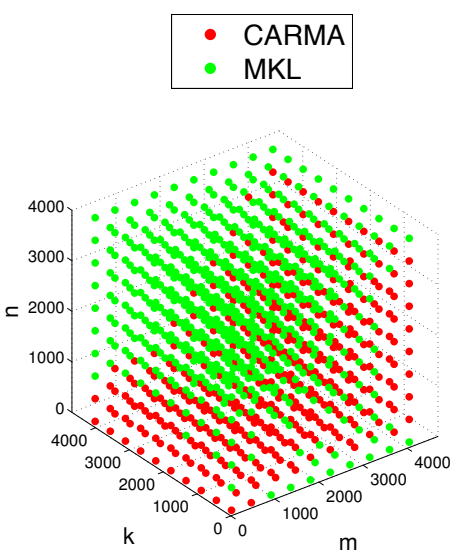
As described in section IV-A, we featurize dense matrix multiplication using the sizes of the input matrices: m, k, and n. We train our classifier on evenly-spaced datapoints within this three-dimensional space.

We view our training data as points within a d-dimensional space, where d is the number of features (or input parameters). In our case, $d = 3$ because our features are the matrix dimensions m, k, and n. Our goal is to train a classifier to identify which regions within the entire d-dimensional parameter space should be solved by which algorithm.

A. SVM Configuration

SVM is a powerful tool for solving pattern classification problems. However, its main drawback in our application is that datapoints in the training data set are all given equal weight in determining the optimal partitioning of the space. Weighted support vector machines (WSVMs) allow the relative importance of datapoints to be taken into account. One application of WSVM is reducing the impact of outliers on the classification rate [14]. We are trying to do the opposite: in our case, an outlier represents an area of the parameter space where one algorithm significantly outperforms the other. We would like to ensure that these areas of extreme performance variation are correctly classified, even if it means misclassifying neighboring regions where performance variation is not as pronounced.

Intuitively, we expect the decision boundary to be highly nonlinear. Therefore, we choose to use support vector machines for classification, as these are known to have good performance for nonlinear classification [2]. Specifically, we use LibSVM's [15] implementation of SVM. We select the Gaussian radial basis function (RBF) kernel [16] due to its high performance on non-linear spaces, and set the RBF constant (gamma) to 1.0^{-6}. We use the standard SVM algorithm C-SVC [2], [17] with a cost parameter of 1.0.

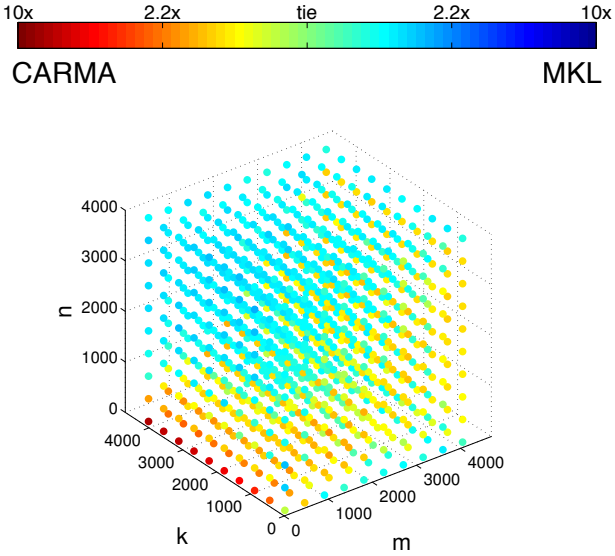


(a) Hopper

(b) Emerald

(c) Boxboro

Fig. 1: Training results for each machine. Green dots represent datapoints where MKL is faster, and red dots represent datapoints where CARMA outperforms MKL.

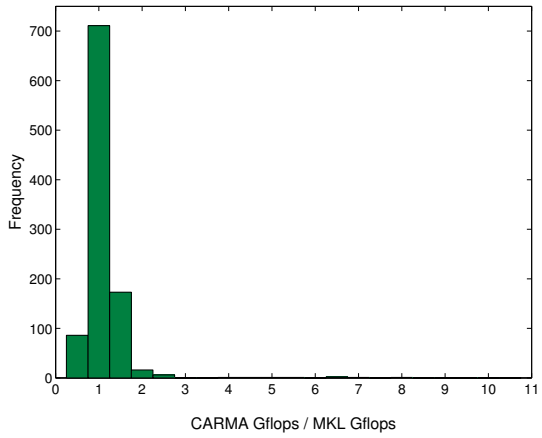


(a) Hopper

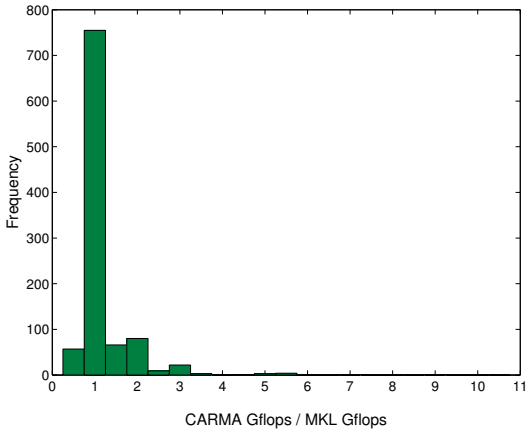
(b) Emerald

(c) Boxboro

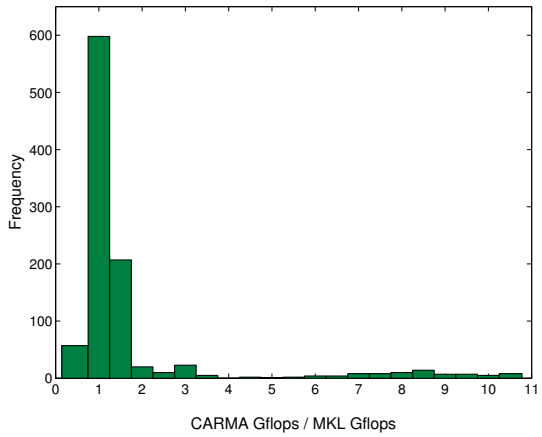
Fig. 2: Heat plots. Color scale is a polynomial function of how much faster one algorithm is than the other (see legend for scale).



(a) Hopper

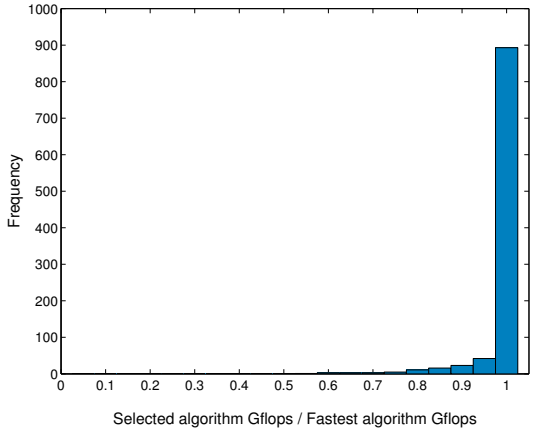


(b) Emerald

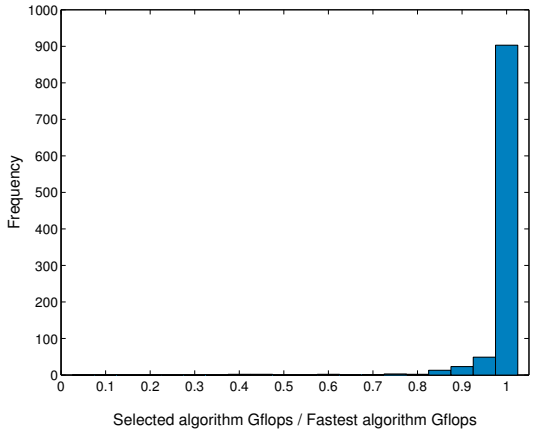


(c) Boxboro

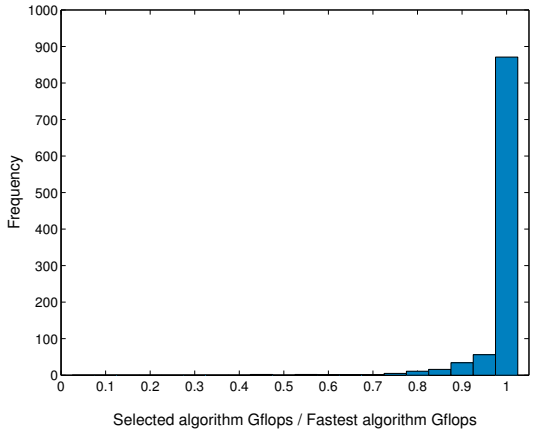
Fig. 3: Histograms of CARMA performance relative to MKL for training data.



(a) Hopper



(b) Emerald



(c) Boxboro

Fig. 4: Histograms of selected algorithm performance relative to the optimal algorithm for test data.

Finally, we set the weight w_i for each class i to 1.0 so that no algorithm is inherently favored over another.

B. Training Data

We trained the classifier on three different machines: Hopper, Emerald, and Boxboro (see Table I). For each machine, we used the maximum number of available cores.

Our results, shown in Figure 1, were surprising. While we expected some machine-to-machine variation, the regions where each algorithm performs better are substantially different across machines. We do not have a theoretical model to explain this variation, which further illustrates the value of our empirical machine learning approach. This is a key result: techniques that do not consider per-machine variation are certain to use the wrong algorithm in many cases. We also note that the regions of optimal performance for each algorithm are in fact nonlinear.

C. Weighting the Data Instances

Some regions of the parameter space may exhibit extreme performance variation. In these cases, it is beneficial to classify those regions correctly at the expense of misclassifying other areas where each algorithm's performance is comparable. In order to address this, we use Weighted SVM (WSVM) [14]. This allows us to assign a weight to each training point. The magnitude of each weight depends linearly on the performance improvement of the faster algorithm relative to the slower algorithm. The lower a training example's weight, the less significant that datapoint is in creating the model.

For a given sample point s, $P_f(s)$ is the performance in Gflops/sec of the faster of the two algorithms and $P_s(s)$ is the performance of the slower algorithm. We choose a scalar constant C, and compute the weight $w(s)$ by the following formula:

$$w(s) = C \cdot \left(\frac{P_f(s)}{P_s(s)} - 1 \right)$$

With this weight function, more of the regions with high performance disparities are correctly classified. We choose $C = 5.0$ (we achieved the best performance with this value) and use the WSVM implementation from the LIBSVM Tools library [15].

Figures 2 and 3 shows the relative performance variation between MKL and CARMA for the entire parameter space. There are clear regions where correct classification improves our performance metric significantly, even if they are small regions that don't drastically affect classifier accuracy. Although there are some apparent performance patterns across machines (namely matrices with large k and small m and n are consistently dominated by CARMA), algorithm selection is not trivial in most regions.

TABLE II: Classifier Accuracy.

Machine	Accuracy
Hopper	86.8%
Emerald	87.0%
Boxboro	84.6%

VI. Classifying

To evaluate our classifier, we randomly generate values for m, k, and n within our training space. For each of the three machines, we generate 1000 datapoints (a datapoint being a (m, k, n) tuple) and replicate our data-collecting procedure to time the multiplications. We use this data as the basis to evaluate our classifier. For each test datapoint, we use the SVM model for the specific machine to generate the classifier's prediction. Given the predicted labels and the true algorithm performance measurements, we are ready to evaluate our classifier.

VII. Evaluation

We evaluate classifier performance as well as selection performance. The former shows how accurate our model is, while the latter shows how valuable our model is for improving overall performance.

A. Classifier Performance

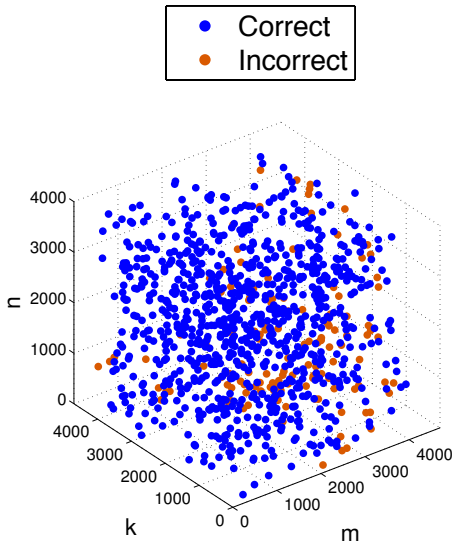
To evaluate the classifier's performance, we measure its accuracy. We simply divide the number of correctly classified test datapoints by the total number of test datapoints. The classifier achieved above 84% accuracy on each of our machines. See Table II for complete accuracy results and Figures 5 and 4 for a visual representation of the test data.

B. Selection Performance

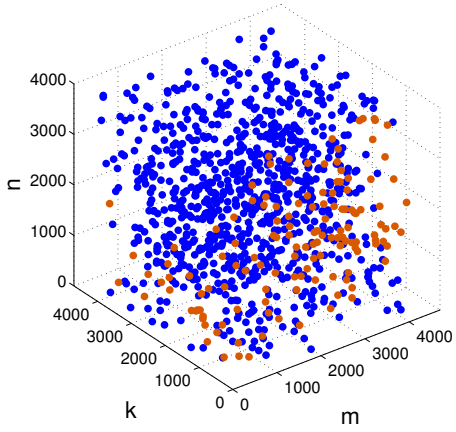
We also performed tests to quantify the benefit of using our algorithm selection technique. The question we wish to answer is: how much performance do we gain by using our classifier to select the optimal algorithm? The answer depends on the machine, as well as what algorithm would have been chosen otherwise.

In order to quantify the value of using our algorithm selector, we calculated four statistics based on the test data. *Selection vs CARMA Only* and *Selection vs MKL Only* measure the average percent improvement of using the selector to choose an algorithm versus always choosing CARMA or MKL, respectively. These statistics include the penalty of incorrect algorithm selections, and provide insight into the value of using our classifier for algorithm selection.

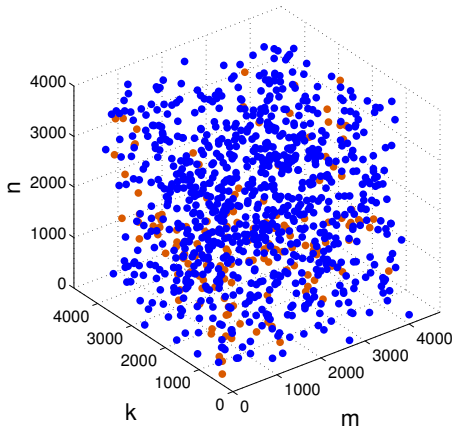
We also compare our technique to both the best and worst case scenarios. *Gain Over Worst* measures the average gain in performance of our classifier over the worst possible algorithm at each datapoint. Similarly, *Loss Under Best* measures how far our classifier is from a 100%



(a) Hopper (86.8% accurate)

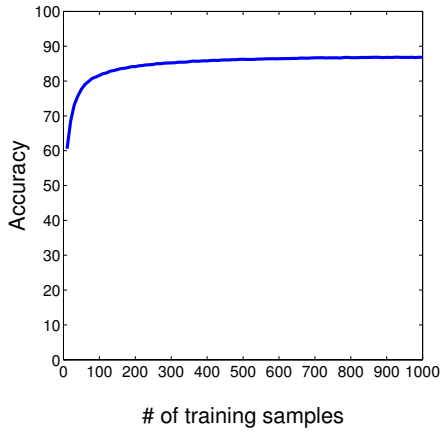


(b) Emerald (87.0% accurate)

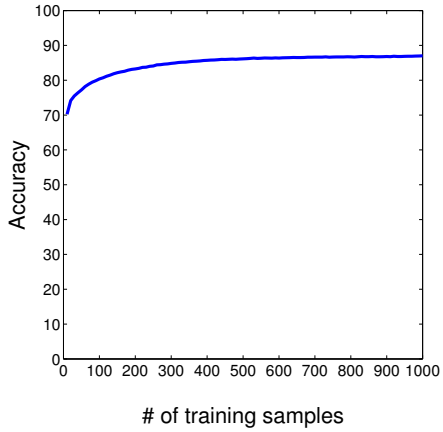


(c) Boxboro (84.6% accurate)

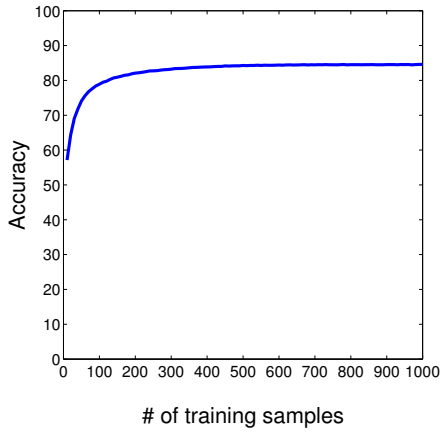
Fig. 5: Testing results for each machine. Blue dots represent datapoints where our classifier correctly predicts which algorithm was faster, and orange dots represent incorrect classifications.



(a) Hopper



(b) Emerald



(c) Boxboro

Fig. 6: Convergence plots for each machine. High accuracy is achieved with very few training samples.

ideal algorithm selector: it simply averages the percent loss of our classification versus the correct classification for all data points.

The highest *Selection vs CARMA Only* is 10.8% (on Hopper), and the highest *Selection vs MKL Only* is 25.7% (on Boxboro). Boxboro also yields a *Selection vs CARMA Only* of 9.1%, which shows that one cannot simply choose the "better on average" algorithm and always expect to achieve near-optimal performance across the parameter space. In fact, all three machines show relatively "balanced" performance in the sense that neither CARMA nor MKL dominate the majority of the parameter space. These are the cases in which our algorithm selection technique provides the greatest value. All of these statistics are shown in detail in Table III.

The average performance loss resulting from an incorrect classification ranges from 12% to 14%, indicating that the majority of misclassifications occur in areas where CARMA's and MKL's performance are comparable. By comparison, the *Average Correct Classification Gain* (i.e. the average performance gain of correctly classified datapoints) ranges from 27% to 43% (see Table VI). In other words, when we correctly classify a datapoint, the faster algorithm is *on average* 27% to 43% faster than the slower algorithm. Also, the most extreme performance gap is an 11× speedup of CARMA over MKL (on Boxboro). Therefore, *Average Misclassification Error* rates of 12% are relatively low, especially considering that misclassifications occur at only 14% of the test datapoints (accuracy is roughly 86%).

Our algorithm selection technique is very close to optimal: on the three machines tested, *Loss Under Best* never exceeds 1.5% on average. In other words, using our classifier's set of selected algorithms yields at least 98.5% of the maximum performance possible across the set of test data we used. This attests to the value of our classifier; although accuracy does not exceed 87% (see Table II), performance reaches at least 98.5% of optimal. These statistics are shown in detail in Table III.

C. Impact of Weighting Datapoints

Table IV shows the performance gains of our classifier without using weighted datapoints (as described in Section V-C). Comparing Table IV with Table III shows that on all three machines, for all four statistics described in Section VII-B, the weighted SVM classifier outperforms the unweighted version. Therefore, we conclude that using weighted SVM for algorithm selection is superior to using unweighted SVM.

D. Properties of Misclassifications

For any complex performance space (such as ours), we can expect that our algorithm selector will not always choose the optimal algorithm. The data shows this: although we get close to optimal performance (1.1% to 1.5% below optimal, see Table III), we still have some misclassifications. Table V shows *Average Misclassification Error*, which is the average performance penalty of misclassified datapoints[3].

An important observation is that the majority of incorrect classifications occur on or near the decision boundaries our classifier generates. This is due to the fact that along these decision boundaries, the performance of MKL and CARMA is comparable: on any given trial, CARMA may slightly outperform MKL, or MKL may marginally beat CARMA. Therefore, the algorithm that our classifier selects along a decision boundary may not match the given instance of test performance data that we gathered, even though the actual performance difference is relatively minor. These misclassified marginal points explain why classifier accuracy is roughly 86%, while performance is within 1.5% of optimal.

To visualize this phenomenon, we simplify the problem by reducing it to two-dimensional algorithm selection. By only considering datapoints along planes within the three-dimensional parameter space, we are able to train and test a two-dimensional classifier (see Figure 7)[4]. Observe that in the two-dimensional planes that we chose (namely the $k-(m=n)$, $m-(k=n)$, and $n-(m=k)$ planes), incorrect decisions occur near the boundary determined by our SVM classifier.

E. Classifier Convergence

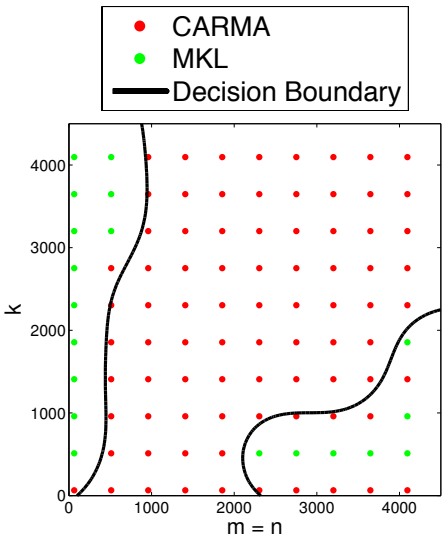
As with all other classifiers, our algorithm selection technique requires the user to specify how many datapoints will be used to train the model. This exposes a tradeoff: classifier accuracy versus training time. The more training datapoints we use, the better the model. However, once our classifier has converged, additional training points do not improve the model and require extra time and resources to generate.

In order to measure how many datapoints are required to train an accurate classifier, we explore the effects of reducing the size of our training set. In this experiment, we randomly select X% of our original 1000 training points, for X ranging from 1% to 100%. After training our classifier with this random X% subset of our training data, we measure the resulting accuracy on the test data. We use accuracy to track classifier convergence (rather than one of the performance metrics described in Section VII-B) because although we are concerned with performance, the classifier is considered "trained" when accuracy has converged.

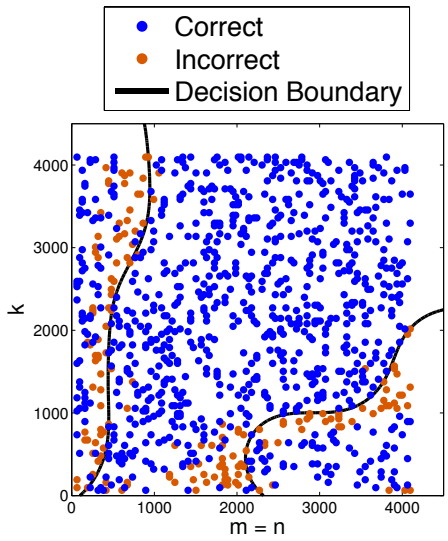
Figure 6 shows these convergence plots for each machine. It is immediately apparent that we need much fewer than 1000 training datapoints to build an accurate

[3] *Average Misclassification Error* only takes into account misclassified datapoints, and does not consider gains of correctly classified datapoints.

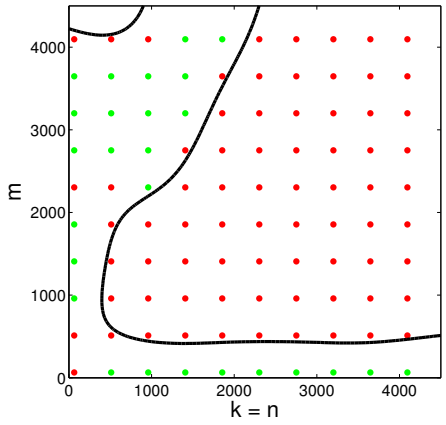
[4] We use MATLAB [18] to generate the SVM model due to MATLAB's built-in ability to show the decision boundary.



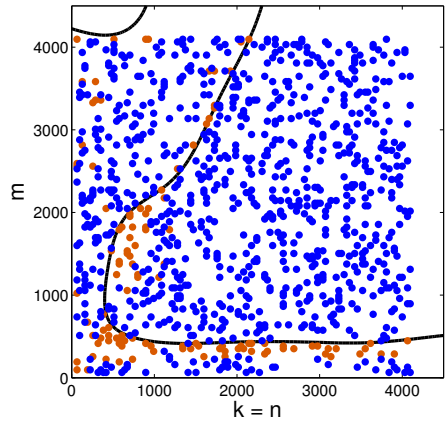
(a) $m = n$, train



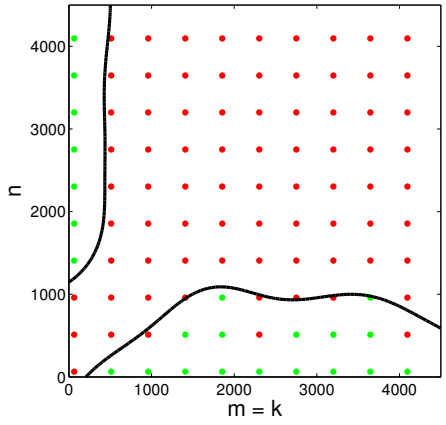
(b) $m = n$, test



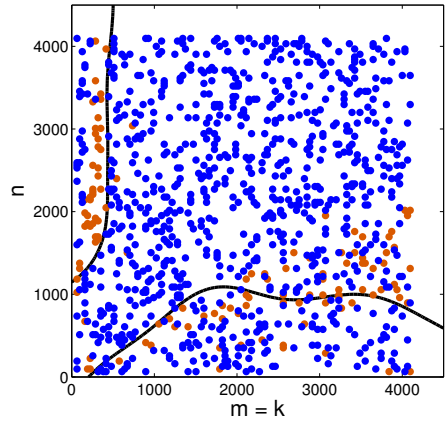
(c) $k = n$, train



(d) $k = n$, test



(e) $m = k$, train



(f) $m = k$, test

Fig. 7: 2D versions (for visualization purposes). All data generated on Emerald. Observe that the majority of incorrect classifications (orange dots in testing plots) occur along the decision boundary, where CARMA and MKL exhibit similar performance behavior.

TABLE III: Runtime performance gains of our algorithm selector.

Machine	Selection vs CARMA Only	Selection vs MKL Only	Gain Over Worst Selection	Loss Under Best Selection
Hopper	10.8%	14.1%	26.3%	-1.4%
Emerald	10.5%	11.7%	23.4%	-1.1%
Boxboro	9.1%	25.7%	36.3%	-1.5%

TABLE IV: Runtime performance gains of our algorithm selector (without weighting datapoints).

Machine	Selection vs CARMA Only	Selection vs MKL Only	Gain Over Worst Selection	Loss Under Best Selection
Hopper	9.7%	12.5%	24.8%	-2.5%
Emerald	10.0%	10.7%	22.4%	-1.6%
Boxboro	9.0%	25.5%	36.1%	-1.7%

TABLE V: Average performance penalty of misclassified data points (does not consider correctly classified datapoints).

Machine	Average Misclassification Error
Hopper	13.7%
Emerald	12.3%
Boxboro	12.7%

TABLE VI: Average performance gain of correctly classified data points (does not consider misclassified datapoints).

Machine	Average Correct Classification Gain
Hopper	30.4%
Emerald	26.8%
Boxboro	42.9%

model. Of course, the actual number of training points required will vary by machine and algorithm. For example, Boxboro requires roughly 100 datapoints to build an accurate model, whereas Hopper and Emerald require 150-200 datapoints to converge. This is due to the fact that performance patterns on Boxboro are less complex. We don't have a theoretical model to understand this complexity, further validating the value our empirical approach.

VIII. Future Work

In this project, we examined dense matrix multiplication on three shared-memory machines. However, the approach of using SVM to partition a feature space based on algorithm performance patterns can be generalized to a wide variety of other algorithms and architectures. We aim to evaluate additional linear algebra algorithms such as QR decomposition and sparse matrix multiplication. Furthermore, we are interested in gathering data on distributed-memory machines in which higher communication costs magnify the performance gains that can be achieved by communication-avoiding algorithms such as CARMA.

In addition to exploring algorithms for problems other than matrix multiplication, we aim to study the effect of a higher-dimensional feature space on the performance of our algorithm selection methodology. Moreover, we are interested in evaluating our approach on a set of more than two algorithms for solving a particular problem. Our current weighting system does not generalize to sets of three or more algorithms. We seek to maximize the number of cases in which the selected algorithm offers significantly better performance than the alternative algorithms. On the other hand, correctly classifying datapoints in which the algorithms' exhibit comparable performance is lower priority. When there are only two algorithms, the formula must only account for the relative performance of algorithm A versus algorithm B. In the three algorithm case, one must account for the performance of algorithm A versus algorithm B, algorithm B versus algorithm C, and algorithm A versus algorithm C. It would be necessary to devise a more general weighting formula that ensures that the optimal algorithm is selected when the impact on performance is greatest.

IX. Conclusion

We've tackled the problem of algorithm selection for HPC and machine learning problems, specifically focusing on dense matrix multiplication. This problem is challenging due to the complexity of analyzing machine performance a priori and due to per-machine variation in algorithm performance, the latter of which was dramatically reflected in our training results. We found that using SVM for algorithm selection can be valuable, with performance

improvements of up to 26% over preselecting a single algorithm. Moreover, achieving that level of accuracy requires a small training set—only a few hundred data points in our experiments.

We believe this technique has promise for improving the performance for a variety real applications. The status quo requires developers to think carefully about the machines and data their applications will use, analyze the expected performance, and then choose their algorithm appropriately, hoping they've made the right choice for all cases. This need not be the case: we've demonstrated that our approach is feasible and in general improves performance, with up to 26% increase in performance in the best case, and within 1.5% of optimal algorithm selection.

X. Acknowledgments

Shaddi Hasan contributed to the design and implementation of the SVM approach to algorithm selection. Philip Reames made the initial recommendation to explore the SVM approach and Dr. John Canny assisted in its development. Orianna DeMasi suggested valuable resources and provided helpful feedback.

Research partially funded by DARPA Award Number HR0011-12-2-0016, the Center for Future Architecture Research (a member of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA), and ASPIRE Lab industrial sponsors and affiliates Intel, Google, Nokia, NVIDIA, and Oracle. Any opinions, findings, conclusions, or recommendations in this paper are solely those of the authors and does not necessarily reflect the position or the policy of the sponsors.

References

[1] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. D. Kubiatowicz, E. A. Lee, N. Morgan, G. Necula, D. A. Patterson *et al.*, "The parallel computing laboratory at UC Berkeley: A research agenda based on the Berkeley view," *EECS Department, University of California, Berkeley, Tech. Rep*, 2008.

[2] C. Cortes and V. Vapnik, "Support-vector networks," *Machine Learning*, vol. 20, no. 3, pp. 273–297, 1995.

[3] K. Leyton-Brown, E. Nudelman, G. Andrew, J. McFadden, and Y. Shoham, "A portfolio approach to algorithm selection," in *IJCAI*, vol. 1543, 2003, p. 2003.

[4] Intel, "Math Kernel Library," software.intel.com/en-us/intel-mkl.

[5] J. Demmel, D. Eliahu, A. Fox, S. Kamil, B. Lipshitz, O. Schwartz, and O. Spillinger, "Communication-Optimal Parallel Recursive Rectangular Matrix Multiplication," in *IEEE International Parallel & Distributed Processing Symposium*, 2013.

[6] J. R. Rice, "The algorithm selection problem," *Advances in Computers*, vol. 15, pp. 65–118, 1976.

[7] J. Bergstra, N. Pinto, and D. Cox, "Machine learning for predictive auto-tuning with boosted regression trees," in *Innovative Parallel Computing (InPar), 2012.* IEEE, 2012, pp. 1–9.

[8] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Satzilla: Portfolio-based algorithm selection for SAT." *J. Artif. Intell. Res.(JAIR)*, vol. 32, pp. 565–606, 2008.

[9] M. G. Lagoudakis and M. L. Littman, "Algorithm selection using reinforcement learning." in *ICML*, 2000, pp. 511–518.

[10] R. Vuduc, J. W. Demmel, and J. Bilmes, "Statistical models for automatic performance tuning," in *Computational Science—ICCS 2001.* Springer, 2001, pp. 117–126.

[11] R. Vuduc, J. W. Demmel, and J. A. Bilmes, "Statistical models for empirical search-based performance tuning," *International Journal of High Performance Computing Applications*, vol. 18, no. 1, pp. 65–94, 2004.

[12] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe, "Petabricks: A language and compiler for algorithmic choice," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Dublin, Ireland, Jun 2009. [Online]. Available: http://groups.csail.mit.edu/commit/papers/2009/ansel-pldi09.pdf

[13] J. Ansel, S. Kamil, K. Veeramachaneni, U.-M. O'Reilly, and S. Amarasinghe, "OpenTuner: An extensible framework for program autotuning," 2013.

[14] X. Yang, Q. Song, and Y. Wang, "A weighted support vector machine for data classification," *International Journal of Pattern Recognition and Artificial Intelligence*, vol. 21, no. 05, pp. 961–976, 2007.

[15] C.-C. Chang and C.-J. Lin, "LIBSVM: A library for support vector machines," *ACM Transactions on Intelligent Systems and Technology*, vol. 2, pp. 27:1–27:27, 2011, software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm.

[16] M. D. Buhmann, *Radial basis functions: theory and implementations.* Cambridge University Press, 2003, vol. 12.

[17] B. E. Boser, I. M. Guyon, and V. N. Vapnik, "A training algorithm for optimal margin classifiers," in *Proceedings of the fifth annual workshop on Computational learning theory.* ACM, 1992, pp. 144–152.

[18] MATLAB, *version 7.14.0 (R2011b).* Natick, Massachusetts: The MathWorks Inc., 2011.